A Hibernation Aware Dynamic Scheduler for Cloud Environments

Luan Teylo, Luciana Arantes, Pierre Sens, Lúcia Maria de A. Drummond

A Hibernation Aware Dynamic Scheduler for Cloud Environments

Luan Teylo
Fluminense Federal University
Niteroi, Brazil
luanteylo@ic.uff.br

Luciana Arantes
Sorbonne Université, CNRS, INRIA, LIP6
Paris, France
luciana.arantes@lip6.fr

Pierre Sens
Sorbonne Université, CNRS, INRIA, LIP6
Paris, France
pierre.sens@lip6.fr

Lúcia Maria de A. Drummond
Fluminense Federal University
Niteroi, Brazil
lucia@ic.uff.br

ABSTRACT

Nowadays, cloud platforms usually offer several types of Virtual Machines (VMs) which have different guarantees in terms of availability and volatility, provisioning the same resource through multiple pricing models. For instance, in the Amazon EC2 cloud, the user pays per hour for on-demand VMs while spot VMs are unused instances available for a lower price. Despite the monetary advantages, a spot VM can be terminated or hibernated by EC2 at any moment. In this work, we propose the Hibernation-Aware Dynamic Scheduler (HADS), to schedule applications composed of independent tasks (bag-of-tasks) with deadline constraints in both hibernation-prone spot VMs (for cost sake) and on-demand VMs. We also consider the problem of temporal failures, that occurs when a spot VM hibernates, and does not resume within a time that guarantees the application's deadline. Our dynamic scheduling approach aims at minimizing the monetary costs of bag-of-tasks applications execution, respecting its deadline even in the presence of hibernation. It is also able to avoid temporal failures, by using task migration and work-stealing techniques. Experimental results with real executions using Amazon EC2 VMs confirm the effectiveness of our scheduling when compared with on-demand VM only based approaches, in terms of monetary costs and execution times. It is also shown that our strategy can tolerate temporal failures.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Availability**; **Maintainability and maintenance**.

KEYWORDS

Clouds, Temporal failures, Hibernation, Dynamic Scheduling

1 INTRODUCTION

In the past few years, cloud computing has emerged as an attractive option to run different classes of applications due to several advantages over other platforms, such as: (i) immediate access to computational resources, (ii) no upfront capital investments, and (iii) pay-per-use model. Some cloud providers offer several classes of Virtual Machines (VMs) with different guarantees in terms of availability and volatility, provisioning the same resource through multiple pricing models. Amazon EC2, for example, offers VMs in two main markets: the on-demand and the spot markets.

On-demand VMs do not require a long-term commitment and can be deployed at any time. Those VMs have high availability and cannot be interrupted by the provider. On the other hand, spot VMs are unused EC2 capacity offered by Amazon with a huge discount (according to Amazon it can be up to a 90% discount compared to on-demand prices). But it can be interrupted by the provider when it needs the resources back.

In December 2017, Amazon EC2 changed the spot VMs pricing model. Before that, to contract a spot VM the user had to give a bid and the VM was allocated only if its current price was smaller than that bid. If during the execution, the VM price increased and became larger than the bid, the VM was interrupted. Before the new price model, the spot market prices faced constant variations because it was driven by the users' bids. With the new price model, the client does not have to give a bid anymore to contract spot VMs, and the prices are stable. In addition to prices stability, the number of interruptions decreased and Amazon EC2 also included the hibernation feature on the spot VMs. So, now the provider can hibernate a spot VM instead of terminating that. When a spot VM is hibernated, its memory and context are saved and, when the cloud demand reduces, the VM is resumed by the provider, and all the tasks that were executing at the moment of the hibernation restart from the breakpoint. Note that the user is not charged for the periods of the hibernation.

In this paper, we are interested in applications composed of independent tasks which can be executed in any order and in parallel, called bag-of-task (BoT). Although simple, the BoT approach is used by several applications such as parameter sweep, chromosome mapping, Monte Carlo simulation and computer imaging applications. Furthermore, they may require deadline-bounds where the correctness on the computation also depends on the time the computation of all tasks ends.

In this work, we propose a dynamic cloud scheduler for Bag-of-Tasks applications using, for cost sake, hibernation-prone spot VMs as much as possible, respecting the application deadline constraint while also minimizing the monetary costs. Our strategy is able to avoid the problem of temporal failures, that occurs when a spot VM hibernates, and does not resume within a time that guarantees the application's deadline, by using task migration and work-stealing techniques. Experimental results with real executions using Amazon EC2 VMs confirm the effectiveness of our scheduling when compared with on-demand VM only based approaches, in terms

of monetary costs and execution times. It is also shown that our strategy can tolerate temporal failures.

So, the main contributions of this work are:

- Definition of the BoT scheduling problem to hibernation-prone spot VMs
- Development of the Hibernation-aware Dynamic Scheduler (HADS)
- Evaluation of HADS in different scenarios of VMs hibernation and resuming

The remaining of the paper is organized as follows. Section 2 discusses some related work. Section 3 presents the problem definition, and Section 4 describes our proposed dynamic scheduler. Evaluation results conducted in real scenarios are presented in Section 5. Finally, Section 6 concludes the paper and introduces some future directions.

2 RELATED WORK

According to Oprescu and Kielmann [10], applications composed of independent tasks lend themselves well to elastic environments, where computational resources can be added or removed according to the application's needs. Because of that, many works propose scheduling BoT applications on homogeneous and heterogeneous cloud environments [15]. For instance, Thai et al. [14] evaluate the scheduling of applications in on-demand VMs distributed across different data centers, focusing on the trade-offs between performance and cost, while Yao et al. [18] provide an approach that satisfies job deadlines and minimizes monetary cost. The proposed heuristics use both on-demand and reserved VMs. In [15], Thai et al. present an extensive survey and taxonomy of existing research in the scheduling of bag-of-task applications in clouds.

Gutierrez-Garcia and Sim [8] present an agent-based strategy that uses different scheduling heuristics to schedule concurrent BoTs applications to virtual machines in the cloud. In [10], the authors propose BaTs, a budget-constrained scheduler that uses on-demand VMs to execute BoT applications. Farahadaby et al. [6] propose FPRAS, a scheduler algorithm for BoT applications in multi-cloud environments. In the same line, in [16] a binary integer program to select computational resources in different cloud providers is presented. This work uses a static approach to scheduling non-finite applications whose main objective is to maximize the computational capacity of the infrastructure, respecting a budget constraint. Unlike our approach, those works do not consider the use of VMs spots to minimize the monetary cost of the execution and do not employ techniques to guarantee the continuity of the applications in case of VMs interruptions.

Checkpointing is widely used to tolerate failures in different environments. Checkpointing [3, 7] approaches periodically save the execution state of the application as an image file. These mechanisms can be costly to the cloud since data centers have limited network resources and may readily become overloaded when a huge number of checkpoint image files need to be stored. Moreover, they are usually not suitable for real-time applications or jobs with deadline constraints since periodically checkpoints and resuming of failed tasks are time-consuming.

Replication and resubmission of tasks are other mechanisms widely used to tolerate failures [11, 17, 19]. Using replication, several copies of the same task are executed to support fault tolerance. Most of the studies use a single primary-backup scheme considering one primary and one or several backup (copy) tasks scheduled in different computing instances [17, 19]. A backup is executed when its primary cannot complete execution due to failure but it does not require fault diagnosis. Zheng et al. [19] propose an algorithm to find an optimal backup schedule for each independent task. Wang et al. [17] extend Zheng's results in a cloud context and using an elastic resource provisioning. Despite the use of overlapping techniques, primary-backup schemes require that tasks have enough time for executing backups in case of failure. [5] presents a comprehensive study of replication schedulers where all replicas of a task start executing concurrently and the next task is started as soon as one of the previous task replicas finishes. Benoit et al. [4] adopt a more conservative approach where the next task can only start when all the replicas of the previous task finished. Contrarily to our approach, the above solutions need to schedule both the primary and backup tasks and the later takes the execution control if the former fails. Neither of them uses backup tasks to avoid temporal failure.

Different mechanisms and tools have been proposed in the literature to deal with the revocation of spot VMs. The most common objective is to maintain a tradeoff between the monetary cost and the reliability of the execution. SpotOn [13] is a batch service computing that uses checkpointing, migration and replication to mitigate the impact of spot VMs revocations. Lu *et al.* use hybrid instances, including both on-demand instances for high priority tasks and backup, and spot instances for normal computational tasks.

In [9] a machine learning algorithm to select spot and on-demand VMs to execute batch jobs that arrive over time is proposed. The solution also switches to on-demand resources when there is no spot instance available to ensure the desired performance. SpotCheck [12] uses nested VMs within spot VMs to provide the illusion of a platform that offers always-available VMs. The nested VMs are migrated to an on-demand VM when a spot revocation is detected. Those works use the price variation of the spot market to predict the spot VMs' revocations. However, with the changes in the spot market announced in December 2017, now all prices are stable, making it no longer possible to predict the termination of the VMs.

To the best of our knowledge, no work that considered the use of spot VMs to execute BoT applications has explored the hibernation-prone spot VMs to minimize the monetary cost of the execution. Besides that, there are no evaluations of the impact of the hibernation in the execution of the applications.

3 PROBLEM DEFINITION

Let M be the user-provided set of VMs that can be used to execute the BoT application, where $M^s \subset M$ is the set of spot instances and $M^o \subset M$ is the set of on-demand ones. Moreover, let D be the deadline defined by the user and $T = \{1, \ldots, D\}$ be a discrete set that represents the periods used during the execution. Each $vm_j \in M$ has a memory capacity of m_j gigabytes, and a set of virtual cores VC_j. Whenever a new VM is allocated it takes α periods to be available, which includes the time to answer the client request and the boot time overhead.

Each VM $vm_j \in M$ is offered in only one of the markets (spot or on-demand) with cost c_j. When a VM is allocated to a user, he/she pays for a full-time interval called Allocation Cycle (AC), which is usually one hour. Thus, if a VM was used for 61 minutes, the user will be charged for two ACs (120 minutes). Note that an AC can correspond to several periods in T. For example, if each period corresponds to one second, an AC of one hour would correspond to 3600 periods.

Let B be the set of tasks of a BoT application. We assume that each task $t_i \in B$ is executed in only one core and requires a known amount of memory rm_i, which must be available throughout its execution. Therefore, a multi-core VM can execute more than one task simultaneously (one task per core) only when there is enough main memory to allocate them. We also consider that the execution time of each task t_i in any $vm_j \in M$ is also known and given by e_{ij}. Figure 1a presents the execution of a set of tasks in a spot VM. As can be seen, after α periods each core starts executing a task. However, note that there is a gap between tasks 4 and 5. That happened because the VM does not have enough memory to fulfill the requirement of tasks 1 and 5 at the same time; thus $core_0$ remains idle until task 1 finishes.

In an environment where VMs are hibernation-prone, a hibernated VM can resume in time to meet the deadline or not. In the first situation, shown in Figure 1b, since the VM resumes in time, tasks go on running from the break-point. On the other hand, if the VM does not resume in time, it is necessary to migrate the affected tasks to different VMs to avoid a temporal failure. If the others already deployed spot VMs were not able to execute the affected tasks in time, new on-demand VMs are deployed.

Figure 1c illustrates a hibernation followed by migration. Considering that the spot VM hibernated in time $p \in T$, $Q_{jp} \subset B$ is the set of tasks to be migrated if vm_j does not resume in time, so it contains both the tasks that were running in vm_j at the moment of hibernation and the tasks that were waiting to be executed in that virtual machine. In Figure 1c, $Q_{jp} = \{1, 2, 4, 5\}$.

We define the set of VMs that will receive the affected tasks by hibernation as $K \subset M$. We also consider that: (i) a VM takes α periods of time to be ready to receive the migrated tasks and (ii) the number of periods required to perform all tasks of Q_{jp} in VMs of K is $rt(Q_{jp}, K)$. Equation 1 calculates the hibernation time limit ($st \in T$), when the migration must be triggered. If the migration does not start at time st, it will not be possible to meet the deadline. Thus, we aim at finding a set of instances $K \subset M$, such that $st + rt(Q_{jp}, K) \leq (D - \alpha)$.

$$st = (D - \alpha) - rt(Q_{jp}, K) \quad (1)$$

4 HIBERNATION-AWARE DYNAMIC SCHEDULER

The Hibernation-Aware Dynamic Scheduler (HADS) aims to respect the application deadline even in the presence of hibernation, while minimizing the monetary cost. HADS contains the following main procedures: i) a Primary Scheduling Heuristic that defines an initial task allocation map (Section 4.2), and ii) a Dynamic Scheduler Module that monitors the VMs and eventually moves tasks among them so that the deadline is respected and the monetary costs minimized (Section 4.3).

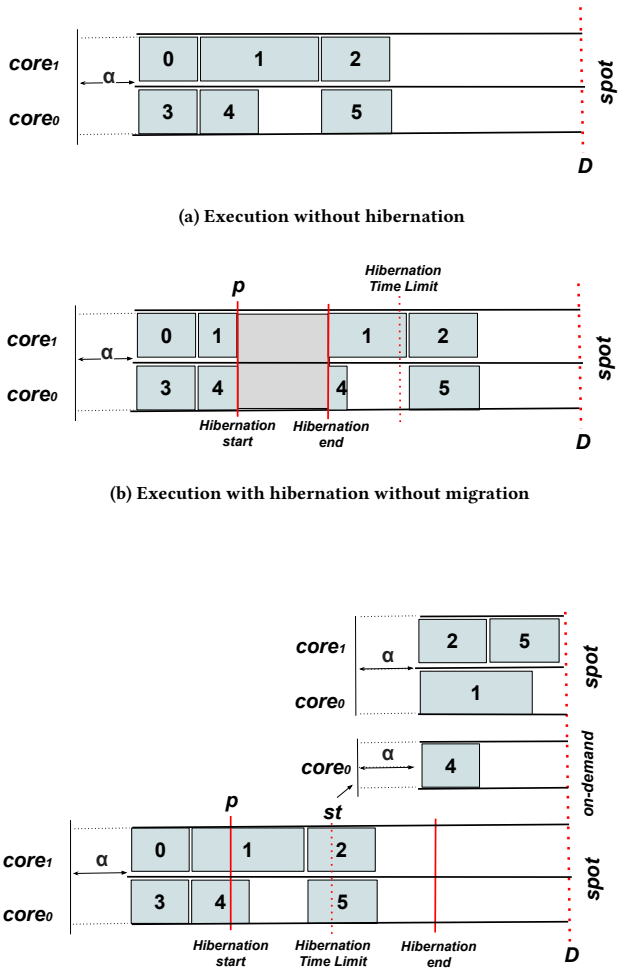


Figure 1: Different scenarios of execution with hibernation-prone spot VMs

4.1 Computing D_{spot}

To schedule the primary tasks, it is necessary to estimate D_{spot}, that is the primary tasks execution times in the worst case. That parameter defines the maximum limit for the primary tasks makespan, and it is used to ensure that in case of a hibernation there will be enough remaining time to migrate the applications to other VMs and finish it before the deadline D.

D_{spot} is estimated using Algorithm 1. In addition to the deadline D and sets B and M, the Algorithm also receives max_spot as input, that is the maximum number of spot VMs that can be allocated simultaneously. This value is defined by the cloud provider. In

Amazon EC2, for example, by default, only 20 VMs spots can be allocated simultaneously in the same region[1].

We can assume that, using Algorithm 2, the number of tasks n allocated to a VM can be estimated as presented in line 1. After that, a set $W \subset B$ containing the n longest tasks is created (line 2). Since W contains the n longest tasks, we can consider that the execution of tasks $t_i \in W$ in the slowest VM of M represents the worst makespan case. In line 7, that makespan, called mkp_w, is estimated using the procedure *get_makespan* by considering the slowest VM selected in line 3 (procedure *get_slowest_vm*). Thus, we calculate the value of D_{spot} as the difference between D and mkp_w (line 8). If D_{spot} is less than zero, the user is warned to consider another deadline.

Algorithm 1 *Compute* D_{spot}

Input: B, M, max_spot, D
1: $n \leftarrow \lceil \frac{|B|}{max\_spot} \rceil$
2: $W \leftarrow get\_longest\_tasks(n, B)$
3: $vmf \leftarrow get\_slowest\_vm(M)$
4: **for all** $ti \in W$ **do**
5: $\quad allocate(t_i, vmf)$
6: **end for**
7: $mkp_w = get\_makespan(vmf)$
8: $D_{spot} = D - mkp_w$

4.2 Primary Scheduling Heuristic

Table 1: Variables of Primary Task Scheduling Heuristic

Name	Description
B	Set of tasks
M^s	Set of VMs spots
D_{spot}	Parameter that determines the maximum occupation period of a spot VM
A	Set of VMs selected to execute primary tasks
t_i	Task t_i
$allocated$	Boolean variable that indicates whether task t_i was successfully scheduled
vm_j, vm_k	Virtual machines

Algorithm 2 shows the primary scheduling heuristic which is a greedy algorithm that allocates the set of tasks $t_i \in B$ to a set of VMs spot. Tables 1 and 2 present the used variables and functions respectively. The algorithm receives B, M^s and D_{spot} as input parameters.

Initially, tasks are ordered in descending order by the memory size they require (line 1). Then, the algorithm tries to allocate each task in an already allocated virtual machine from set A (lines 5 to 11), since it has enough memory and also ensures that the task insertion will respect D_{spot}.

Allocating tasks in an already allocated VM reduces boot time overhead in comparison of allocating a new VM. However, if such an allocation is not possible, the algorithm must allocate a new VM.

[1]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-limits.html

Table 2: Functions and Procedures of Primary Task Scheduling Heuristic

Name	Description
$sort(B)$	Sorts the set of tasks B in descending order by memory size demand
$check\_allocation(t_i, vm_j, D_{spot})$	Checks if the allocation of task t_i to a core of vm_j respects the D_{spot} limit and if vm_j has available memory to meet the requirement of the task (rm_i)
$allocate(t_i, vm_j)$	Allocates task t_i to a core of vm_j
$best\_VM(t_i, M^s)$	Selects the spot VM that executes task t_i with the minimum number of periods of time
$create\_primary\_map(A)$	Creates the initial execution plan

In this case, the heuristic defines the best type of VM in terms of execution time (line 13). Finally, it updates the primary scheduling map (line 15).

Algorithm 2 *Primary Task Scheduling*

Input: B, M^s, D_{spot}
1: $sort(B)$ {Sort tasks by rm_i}
2: $A \leftarrow \emptyset$ {set of allocated VMs}
3: **for all** $ti \in B$ **do**
4: $\quad allocated \leftarrow False$
$\quad$ {Check if vm_j has sufficient time and memory to execute t_i without violating the limit D_{spot}}
5: $\quad$ **for all** $vm_j \in A$ **do**
6: $\quad\quad$ **if** $check\_allocation(t_i, vm_j, D_{spot})$ **then**
7: $\quad\quad\quad allocate(t_i, vm_j)$
8: $\quad\quad\quad allocated \leftarrow True$
9: $\quad\quad\quad stop\_loop()$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad$ **if not** $allocated$ **then**
13: $\quad\quad vm_k \leftarrow best\_VM(t_i, M^s)$
14: $\quad\quad allocate(t_i, vm_k)$
15: $\quad\quad A \leftarrow A \cup \{vm_k\}$ {Update the set of allocated VMs}
16: $\quad\quad M^s \leftarrow M^s - \{vm_j\}$
17: $\quad$ **end if**
18: **end for**
19: $create\_primary\_map(A)$

4.3 Dynamic Scheduler Module

The Dynamic Scheduler Module of HADS is an event-driven algorithm that performs some actions in response to some events that may occur along the application execution in order to reduce the monetary cost and meet the application deadline. These events and the corresponding executed procedures are shown in Figure 2.

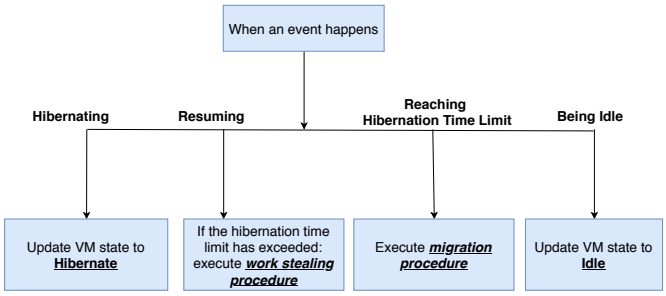


Figure 2: Events and the corresponding procedures of the Dynamic Scheduler Module

We consider that a VM is in one of the following states i) *busy*, if active and executing tasks; ii) *idle*, if active but has finished all tasks allocated to it so far; iii) *hibernated*, if was hibernated by the cloud provider or was not available in the beginning of the application execution; and iv) *terminated*, if was terminated by the Dynamic Scheduler Module.

The events and associated procedures are described in next subsections.

4.3.1 Being idle. When a *busy* VM finishes all tasks allocated to it, its state changes to *idle*. As we will see in Section 4.4, *idle* VMs can return to *busy* states due to the task migration procedure. However, if a VM stays idle for a long time, to avoid additional charges, it is terminated near to the end of the current allocation cycle.

4.3.2 Hibernating. The cloud provider can hibernate *busy* or *idle* spot VMs. In this case, the algorithm changes the VM state to *hibernated*. If the VM was *busy*, the algorithm also computes the hibernation time limit (Equation 1) considering both the tasks that were running as the ones that were waiting for execution when the VM hibernated.

4.3.3 Reaching hibernation time limit. As seen in Section 3, when a hibernated VM that was *busy* reaches the hibernation time limit, in order to meet the deadline all affected tasks are migrated to other VMs. Thus, when this event happens the algorithm starts the migration procedure (Section 4.4).

4.3.4 Resuming. A VM may resume before the hibernation time limit or not. As seen in Section 3, if a VM resumes before the time limit, the tasks continue their executions from the break-points and no additional action is taken. On the other hand, when a VM resumes after the time limit, before it happens, the event *Reaching hibernation time limit* has already happened and consequently the tasks allocated to that VM were already migrated to other VMs. Then, in this case, the algorithm executes a work-stealing procedure that tries to bring back tasks from on-demand VMs to the resumed spot. That procedure is described in Section 4.5.

4.4 Migration Procedure

The migration procedure is presented in Algorithm 3. As can be seen there, the algorithm receives as input the set $Q_{kp} \subset B$, that contains the remaining tasks of a spot $vm_k \in M$ that hibernated at period $p \in T$. Besides that, the procedure also receives the deadline D and the computational resources for where those tasks can be migrated: $IR \subset M$ that contains all *idle* VMs; $BR \subset M$ that includes all *busy* VMs; and $M^o \subset M$ that contains on-demand VMs that can be deployed. Tables 3 and 4 show the used variables and functions, respectively.

Table 3: Variables of Migration Procedure

Name	Description
Q_{kp}	Set of tasks to be migrated
D	Deadline defined by the user
IR	Set of *idle* VMs
BR	Set of *busy* VMs
M^o	Set of on-demand VMs that can be allocated
t_i	A task from set Q_{kp}
vm_j	A VM from set IR, BR or M^o
$start_{ij}$	The period that a task t_i will start if it is allocated to a vm_j
e_{ij}	Execution time of a task t_i in a vm_j
α	Number of periods a VM takes to become available for execution
$migrated$	Boolean variable that indicates whether task t_i was successfully migrated

Table 4: Functions and Procedures used in the Migration Procedure

Name	Description
$sort\_by\_market()$	Sorts the sets IR or BR prioritizing spot VMs over the on-demand ones
$enough\_mem(t_i, vm_j)$	Returns TRUE, if there is enough memory to allocate t_i to vm_j, otherwise returns FALSE
$migrate(t_i, vm_j)$	Migrates task t_i to one of the virtual cores of vm_j
$sort\_by\_price(M^o)$	Sorts the set of on-demand VMs M^o in ascending order by the VM's price
$start\_vm(vm_j)$	Deploys a new on-demand vm_j in the cloud

For each task t_i in Q_{kp}, the algorithm at first tries to allocate t_i to one of the *idle* VMs of set IR (lines 5 to 13). If it is not possible, it tries *busy* VMs of set BR (lines 17 to 23). In the last case, the task is allocated to a new on-demand VM of set M^o (lines 28 to 36). That approach tries to minimize the monetary costs, since the client pays for ACs and the current AC of *idle* and *busy* VMs may have already been charged by the provider.

Note that the algorithm always gives priority to spot VMs, both in the case of *idle* VMs (line 4) and *busy* VMs (line 16) as well, to reduce monetary costs.

To migrate a task t_i to a virtual machine vm_j, the algorithm checks if the start period of t_i in vm_j ($start_{ij}$) plus the corresponding execution time (e_{ij}) is smaller than the deadline. Since *idle* and

Algorithm 3 *Migration Procedure*

Input: Q_{kp}, D, IR, BR, M^o

```
1:  for each t_i ∈ Q_kp do
2:      migrated ← False
3:      {/*Attempt 1*/}
4:      sort_by_market(IR) {/* Prioritizes spot VMs */}
5:      for each vm_j ∈ IR do
6:          if start_ij + e_ij < D and enough_mem(t_i, vm_j) then
7:              migrate(t_i, vm_j)
8:              IR ← IR − {vm_j}
9:              BR ← BR ∪ {vm_j}
10:             migrated ← True
11:             stop_loop()
12:         end if
13:     end for
14:     if migrated is False then
15:         {/*Attempt 2*/}
16:         sort_by_market(BR) {/* Prioritizes spot VMs */}
17:         for each vm_j ∈ BR do
18:             if start_ij + e_ij < D and enough_mem(t_i, vm_j) then
19:                 migrate(t_i, vm_j)
20:                 migrated ← True
21:                 stop_loop()
22:             end if
23:         end for
24:     end if
25:     if migrated is False then
26:         {/*Attempt 3*/}
27:         sort_by_price(M^o)
28:         for each vm_j ∈ M^o do
29:             if start_ij+e_ij < (D−α) and enough_mem(t_i, vm_j) then
30:                 start_vm(vm_j)
31:                 migrate(t_i, vm_j)
32:                 M^o ← M^o − {vm_j}
33:                 BR ← BR ∪ {vm_j}
34:                 stop_loop()
35:             end if
36:         end for
37:     end if
38: end for
```

busy VMs are already active, the overhead α is not taken into account in these cases (lines 6 and 18). But when a new VM has to be deployed (line 30), the overhead α has to be considered not to violate the deadline(line 29).

When a valid VM is found, the procedure updates the VM state. That is shown in lines 7 and 11, where the selected *idle* VM is removed from set IR and included in the set BR; and from line 31 to 34, where the on-demand VM is removed from set M^o.

4.5 Work-Stealing Procedure

The work-stealing procedure, presented in Algorithm 4, is used to reduce the number of contracted ACs of on-demand VMs, and is executed when a spot VM resumes after the hibernation time limit is reached. When this occurs, the algorithm tries to migrate tasks from on-demand VMs to the resumed VM. All used variables and functions are showed in Tables 5 and 6, respectively.

The Algorithm receives as input the sets of *busy* and *idle* VMs ($BR \subset M$ and $IR \subset M$, respectively), the resumed spot $vm_k \in M$,

Table 5: Variables of Work-Stealing Procedure

Name	Description
BR	Set of *busy* VMs
IR	Set of *idle* VMs
vm_k	A resumed spot VM
D	Deadline defined by the user
vm_j	A *busy* VM from set BR
W	A set of tasks that can be stolen from vm_j
t_i	A task from set W
$start_{ik}$	The period that a task t_i will start if it is allocated to the spot vm_k
e_{ik}	Execution time of a task t_i in the spot vm_k

Table 6: Functions and Procedures used by the Work-Stealing Procedure

Name	Description
$selectTasks(vm_j)$	Returns all tasks that can be stolen from an on-demand vm_j
$migrate(t_i, vm_k)$	Migrates task t_i to one of the virtual cores of the spot vm_k

and deadline D. For each on-demand vm_j in BR, the procedure selects the tasks that can be stolen from vm_j (line 3) and tries to migrate them to vm_k (lines 4-8).

The task selection considers only tasks that are not running and that will not start in the current AC. Figure 3 shows an example of that selection. In this example, tasks are spread over three ACs (AC_1, AC_2, and AC_3). Since the current Allocation Cycle is AC_1, tasks 5, 6, 7 and 8 are candidates to be stolen and start in the next cycles.

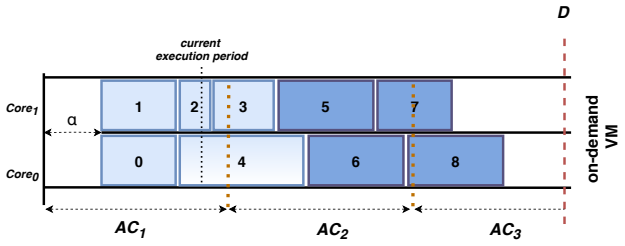


Figure 3: Example of tasks in an on-demand VM that can be stolen by the work-stealing procedure

As in the migration procedure, for each selected task the work-stealing procedure also verifies if the migration would result in the deadline violation, as can be seen in line 5, where the algorithm checks if the start time of the task in the spot vm_k ($start_{ik}$) plus its execution time e_{ik} is smaller than the deadline. If the deadline is not violated, the task is migrated (line 6). Finally, if any task is migrated to the spot vm_k, its state is changed to *busy*, see line 12, and it is included in the set of *busy* VMs. Otherwise, it is updated with *idle* state in line 14.

Algorithm 4 *Work-Stealing Procedure*

Input: BR, IR, vm_k, D

for each $vm_j \in BR$ **do**
  if vm_j **is** on-demand **then**
    $W \leftarrow selectTasks(vm_j)$ {/* get tasks that can be stolen */}
    for each $t_i \in W$ **do**
      if $start_{ik} + e_{ik} < D$ and $enough\_mem(t_i, vm_j)$ **then**
        $migrate(t_i, vm_k)$
      end if
    end for
  end if
end for
if any task was stolen then
  $BR \leftarrow BR \cup \{vm_k\}$
else
  $IR \leftarrow IR \cup \{vm_k\}$
end if

5 EXPERIMENTAL RESULTS

This section presents execution times and monetary costs of different BoT applications executed in the Amazon EC2 virtual machines.

According to the information of Amazon Web Server (AWS)[2], only the VMs of families C3, C4, C5, M4, M5, R3, and R4 with less than 100 GB of memory, running in the spot market, are hibernation-prone. Therefore, for the purposes of this work, we chose to use the third and fourth generation VMs of families C3 and C4. Those VMs are optimized for computation and have high availability in the spot market [3]. Table 7 shows the computational characteristics of the used VMs with its prices in on-demand and spot markets in April 2019.

Table 7: VMs attributes

Type	#VCPUs	Memory	On-demand price per AC	Spot price per AC
c3.large	2	3.75 GB	0.105$	0.0294$
c4.large	2	3.75 GB	0.100$	0.0308$
c3.xlarge	4	7.50 GB	0.210$	0.0588$
c4.xlarge	4	7.50 GB	0.199$	0.0617$

The jobs used in our tests are composed of synthetic tasks generated with the application template proposed by Alves et al. [2]. That application template is based on vector operations, and the execution time depends on the size of the used vectors. Thus, we created several synthetic tasks, each one with memory footprint between 2.81 MB and 13.19 MB, which resulted in execution times varying from 1:42 to 5:30 minutes, as can be seen in Table 8. After that, we created three BoT applications by randomly selecting those tasks.

Table 8: Jobs characteristics

job	# tasks	runtime (minutes)			memory footprint		
		min	avg	max	min	avg	max
J551	60	01:42	03:18	05:23	2.85MB	4.69MB	12.20MB
J552	80	01:43	03:19	05:22	2.91MB	4.71MB	13.19MB
J553	100	01:47	03:10	05:30	2.81MB	4.49MB	10.86MB

Since the provider determines the hibernation and resuming times of the VMs based on internal factors of the cloud such as demand variation, the client cannot control that process. Thus, to evaluate different patterns of VMs hibernation and resuming, we developed a Hibernation Emulation Module (HEM) that emulates the hibernation process, using Poisson distribution [1] to model the hibernation and resuming times.

In our tests, when the hibernation (emulated) occurs, the VM state is saved by using the checkpoint tool CRIU[4], and all tasks allocated to it are paused. In this way, if later the VM resumes, those tasks can be recovered and continue their execution from the checkpoint. Note that HEM uses distinct Poisson functions for the hibernation and resuming modeling, which allows the creation of scenarios where the *hibernating* and *resuming* events have different probability mass functions defined by the parameters λ_h and λ_r, respectively.

To verify the impact of hibernating and resuming events on the monetary cost and execution times, in different scenarios, we considered allocation cycles of 15 minutes ($AC = 900s$) and a fixed deadline of 35 minutes ($D = 2100s$) for all tests. We also considered the VM allocation overhead of 3 minutes ($\alpha = 180s$, based on empirical tests) and the sets M^s and M^o were built considering the allocation limits specified by Amazon EC2 [5]. So, up to five VMs of each type in each market could be allocated.

Let the λ parameter of Poisson distribution be the number of expected events divided by a time interval. Since in our experiments we considered the deadline D as the time interval and variables k_h and k_r, as the expected number of hibernating and resuming events, respectively, λ_h and λ_r parameters are given by $\lambda_h = k_h/D$ and $\lambda_r = k_r/D$.

In the initial tests, we consider that a VM may hibernate only once. So, in practice the values for k_h and k_r determine the chance of a VM hibernation and resuming, respectively. Table 9 presents five created scenarios, considering the minimal values for hibernating and resuming events, as $k_h = 1$ and $k_r = 0$, respectively. So, we consider that at least one hibernation can occur during an execution ($k_h = 1$), but a resuming event does not happen when $k_r = 0$. We also considered five as the maximum value to both events ($k_h = k_r = 5$), because these numbers allowed to observe and analyze all developed procedures.

The scenarios presented in Table 9 were generated combining those cases, including the mean case (c_5). Note that all values shown in this section are averages of three executions in each of these scenarios.

[2] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html
[3] https://aws.amazon.com/ec2/spot/instance-advisor/
[4] https://www.criu.org/
[5] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-resource-limits.html

Table 9: Different execution scenarios generated by varying parameters λ_h and λ_r

ID	*hibernating*	*resuming*	λ_h	λ_r
c_1	$k_h = 1$	$k_r = 0$	1/2100	0/2100
c_2	$k_h = 5$	$k_r = 0$	5/2100	0/2100
c_3	$k_h = 1$	$k_r = 5$	1/2100	5/2100
c_4	$k_h = 5$	$k_r = 5$	5/2100	5/2100
c_5	$k_h = 3$	$k_r = 2.5$	3/2100	2.5/2100

Figure 4 shows the average duration of a hibernation in each scenario. As expected, scenarios c1 and c2 present the longest hibernation times (18:47 and 23:50 minutes, respectively), since in these scenarios the resuming event does not occur ($k_r = 0$). The smallest times are seen in c3 and c4 (10:27 and 11:18 minutes, respectively) that present the highest value of resuming ($k_h = 5$), which reduces the average duration of hibernation. On the other hand, in scenario c5 where $k_r = 2.5$, the hibernation time is longer than in c3 and c4 (13:25 minutes).

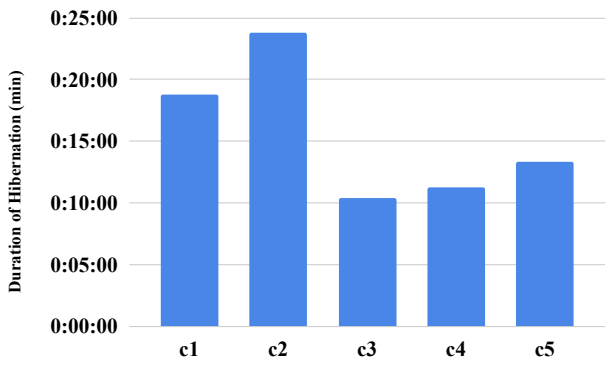


Figure 4: Average duration of hibernation in different scenarios

The results obtained through the execution of jobs in different scenarios presented in Table 9 are discussed in Section 5.1, where the costs were compared with two baseline cases: i) without hibernation, which is the case where the initial scheduling defined by the Algorithm 2 is followed without the need of migration; and ii) on-demand only, which uses the same scheduling, but only with on-demand VMs.

Table 10 presents the average costs of executing the three evaluated jobs (J551, J552, and J553) for the baseline cases. It also presents the number of used VMs, the average makespans in minutes and the percentage difference between their execution costs (diff).

Note that, because the scheduling is the same in both cases, except for the market, the cost difference is around 70% for all executions, which is basically the same difference in the price between the used spots and on-demand VMs (see Table 7). Note also that, in Table 10 the jobs makespans are around 25 minutes instead of the expected 35 minutes of the deadline. This occurs because Algorithm 2 respects the D_{spot} limit presented in Section 4.1.

Table 10: Baseline executions

job	# VMs	makespan	spot without hibernation	on-demand only	diff
J551	3	24:30	0.29$	0.995$	70.75%
J552	4	24:50	0.41$	1.393$	70.57%
J553	4	26:02	0.47$	1.592$	70.55%

5.1 Evaluation of the proposed solution using different test cases.

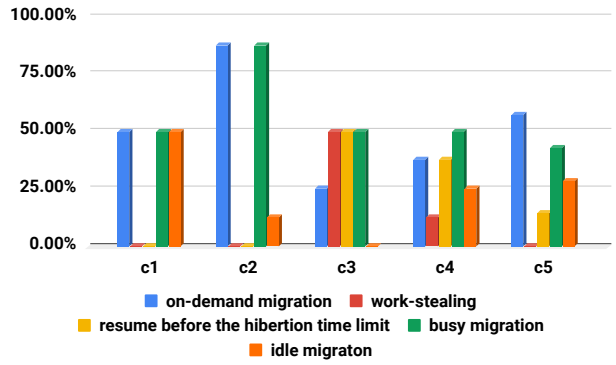
Table 11 presents the execution results of jobs J551, J552 and J553 in each of the evaluated scenarios. It shows the average numbers of hibernations, the number of used on-demand VMs in executions with hibernation, followed by corresponding average values of makespans and the monetary costs. Besides that, the percentage difference between that cost and the cost of the on-demand only approach is presented in the last column (diff). Figure 5 represent the amount of times that a procedure is executed as a percentage of the total number of hibernations occurred along an execution, i.e., for example, if 10 hibernations occur and in 5 of them a work-stealing also happens, the graphic will present a bar with 50%.

Table 11: Average results of three executions of jobs J551, J552 and J553 in different scenarios

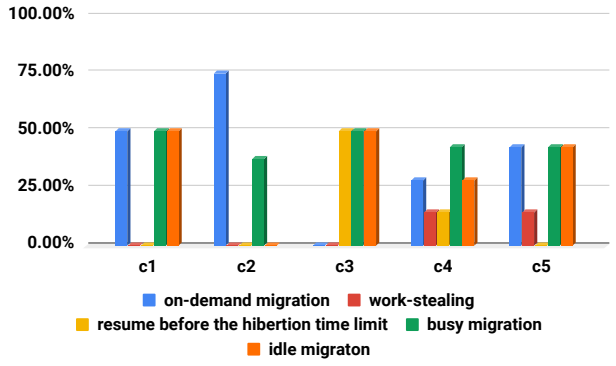
	scenario	# hibernation	# on-demand	makespan (min)	cost	diff
J551 (3 VMs spots)	c_1	0.67	1.00	27:26	$0.373	**62.47%**
	c_2	2.67	4.00	34:02	$0.801	**19.49%**
	c_3	1.33	0.67	28:36	$0.360	**63.85%**
	c_4	2.67	1.33	30:29	$0.538	**45.95%**
	c_5	2.33	2.00	31:30	$0.560	**43.75%**
J552 (4 VMs spots)	c_1	2.00	1.50	32:16	$0.531	**61.89%**
	c_2	4.00	7.50	30:53	$1.550	-11.25%
	c_3	1.00	0.00	24:57	$0.410	**70.55%**
	c_4	3.50	2.00	35:35	$0.727	**47.78%**
	c_5	3.50	3.00	32:06	$0.788	**43.41%**
J553 (4 VMs spots)	c_1	3.00	3.00	32:20	$0.740	**53.55%**
	c_2	3.50	6.00	31:25	$1.380	**13.33%**
	c_3	0.50	0.00	25:31	$0.469	**70.55%**
	c_4	4.00	3.00	35:32	$1.065	**33.08%**
	c_5	3.00	1.50	34:44	$0.698	**56.17%**

As can be seen in Table 11, when compared to the on-demand only approach, the proposed solution presented cost reductions in almost all cases, except only for job J552 in scenario c2 where there was an increase of 11.25% in the cost. In the other cases, the cost reduction varied between 13.33% and 70.55%.

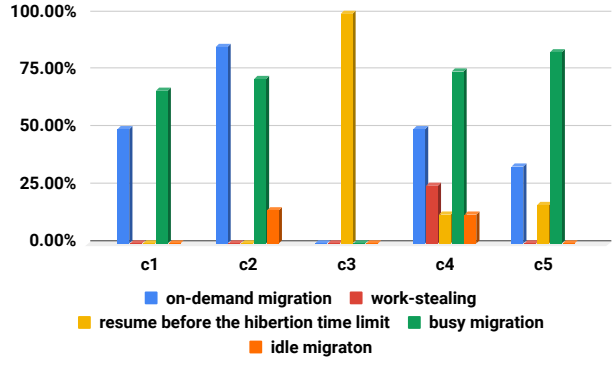
For all jobs evaluated, the worst results are in scenario c2. This is expected since c2 has not VMs resuming ($k_r = 0$) and has the highest hibernation rate ($k_h = 5$). Because of this, in c2 it is always necessary to allocate a large amount of on-demand VMs throughout the execution to avoid temporal failures. Since there is no resuming of spot VMs in this scenario, the work-stealing procedure never executes. Moreover, some tasks initially allocated to the spots had to migrate since the beginning of the execution to on-demand VMs. Particularly, in J552 execution, the spot VMs were hibernated successively, resulting in an allocation of up to eight on-demand VMs and an average increase of 11.25 % in monetary costs. Figure 5 shows that in c2 the migration to on-demand VMs occurs in more than 75% of the hibernation cases.



(a) Job J551



(b) Job J552



(c) Job J553

Figure 5: Percentage distribution of the procedures used by HADS during the execution of jobs J551, J552 and J553

However, note that although there is also no possibility of resuming in scenario c1, the cost reduced in more than 50% for all evaluated jobs. That happens because in this scenario the hibernation rate is low ($k_h = 1$), in general, less than half of the spot VMs hibernates. Thus, the tasks are migrated to busy or idle VMs instead of being allocated to new on-demand VMs. That can be seen in Figure 5, where migrations to idle and busy VMs happen in more than 50% of the hibernation cases.

On the other hand, scenario c3 presents the best results in terms of cost reduction (more than 60% for all jobs). This scenario represents the best execution case because it has the lowest hibernation rate ($k_h = 1$) and the highest resuming rate ($k_r = 5$). Figure 5 shows that spot recoveries occurred in all executions. For example, in J551 and J552 50% of the hibernated VMs resumed before the hibernation time limit, and in J553 this happened in 100% of the hibernated spot VMs.

Comparing scenarios c2 and c4, we see the impact of the resuming event on HADS behavior and the final execution costs. In both scenarios the hibernation rate is $k_h = 5$, where the resuming rate is $k_r = 0$ and $k_r = 5$ to c2 and c4, respectively. Due to that, the cost gain in c4 was more than 30% for all evaluated Jobs, against a maximum gain of 13.33% in c2 (job J553). Thus, although the hibernation rate is the same in both scenarios, we see in Figure 5 that migrations to on-demand VMs occurred in less than 50% in c4, but more than 75% in c2. Also, as the resuming rate is high in c4, we see recoveries, work-stealing, idle and busy migration in all executions.

In c5, which represents the average case ($k_h = 3$ and $k_r = 2.5$), HADS migrates task to on-demand VMs in all evaluated jobs, and the cost reduction was between 30% and 55%. With this, we see that the algorithm's efficiency depends on a balance between the hibernation and resuming rates. Thus, to better understand the impact of these rates, in Section 5.2 we performed several tests varying the frequency of hibernation considering different resuming rates.

5.2 Impact of hibernation and resuming on the execution costs of the jobs

To observe the cost evolution concerning the hibernation rates, we submit job J553, which contains the largest number of tasks, to several scenarios increasing the hibernation rate of each execution progressively. Three cases of tests were considered in this evaluation: i) execution without resuming ($k_r = 0$); ii) execution with medium chance of resuming ($k_r = 3$); and iii) execution with high chance of resuming ($k_r = 7$).

As can be seen in Figure 6, the cost variation appears to have an intrinsic relationship with the resuming rate. The monetary costs approach the on-demand only cost when $k_r = 0$, and distances from it as the rate of resuming event increases.

When $k_h = 6$, all three lines approach to the same cost. At this point, all spot VMs hibernate over different periods of the execution. Some VMs hibernate in the last minutes of the execution, making necessary to allocate on-demand VMs to meet the deadline, regardless of the resuming rate.

However, at the point next to it, the hibernation rate of $k_h = 7$ causes VMs to always hibernate in the first few seconds of the execution. Because of this, in the scenarios where $k_r > 0$, the execution costs are shortly affected, since there is sufficient time to wait for the resuming event. On the other hand, in the case where

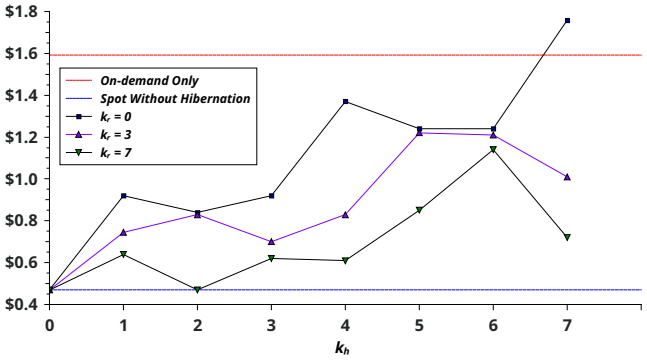


Figure 6: Impact of variation of k_h in the execution costs of job J553

$k_r = 0$, the execution cost is higher than the "on-demand only" approach since the user pays for the first allocation cycle of the spot VMs and for the on-demand VMs used in the migration.

6 CONCLUDING REMARKS AND FUTURE WORK

This paper proposed the Hibernation-Aware Dynamic Scheduler, a dynamic scheduler for bag-of-task applications with deadline constraints that uses both hibernation-prone spot VMs (for cost sake) and on-demand VMs. Our scheduling aims at minimizing monetary costs of bag-of-tasks, respecting application's deadline and avoiding temporal failures. The proposed strategy was evaluated using the VMs of AWS EC2 with real executions of synthetic applications with five scenarios of hibernation. Our results confirmed the effectiveness of our scheduling and that it tolerates temporal failures.

As future work, we intend to test our approach with several real BoT applications and work in a version of the proposed scheduling with checkpoints of the tasks, so that, in the migration case, the tasks can start their executions from the last checkpoints, instead of being re-started from the beginning. Besides that, we also intend to include load balance techniques to increase the occupancy of idle VMs.

ACKNOWLEDGMENTS

This research was supported by *Programa Institucional de Internacionalização* (PrInt) from CAPES (process number 88887.310261/2018-00).

REFERENCES

[1] Joachim H Ahrens and Ulrich Dieter. 1974. Computer methods for sampling from gamma, beta, poisson and bionomial distributions. *Computing* 12, 3 (1974), 223–246.

[2] Maicon Melo Alves and Lúcia Maria de Assumpção Drummond. 2017. A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software* 128 (2017), 150 – 163. https://doi.org/10.1016/j.jss.2017.04.001

[3] Guillaume Aupy, Anne Benoit, Rami G. Melhem, Paul Renaud-Goud, and Yves Robert. 2013. Energy-aware checkpointing of divisible tasks with soft or hard deadlines. In *International Green Computing Conference, IGCC 2013, Arlington, VA, USA, June 27-29, 2013, Proceedings*. 1–8.

[4] Anne Benoit, Mourad Hakem, and Yves Robert. 2008. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. 1–8.

[5] Walfredo Cirne, Francisco Vilar Brasileiro, Daniel Paranhos da Silva, Luís Fabrício Wanderley Góes, and William Voorsluys. 2007. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Comput.* 33, 3 (2007), 213–234.

[6] M Hoseiny Farahabady, Young Choon Lee, and Albert Y Zomaya. 2012. Non-clairvoyant assignment of bag-of-tasks applications across multiple clouds. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 423–428.

[7] Iñigo Goiri, Ferran Julià, Jordi Guitart, and Jordi Torres. 2010. Checkpoint-based fault-tolerant infrastructure for virtualized service providers. In *IEEE/IFIP Network Operations and Management Symposium, NOMS 2010, 19-23 April 2010, Osaka, Japan*. 455–462.

[8] J Octavio Gutierrez-Garcia and Kwang Mong Sim. 2013. A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling. *Future Generation Computer Systems* 29, 7 (2013), 1682–1699.

[9] Ishai Menache, Ohad Shamir, and Navendu Jain. 2014. On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014*. 177–187.

[10] Ana-Maria Oprescu and Thilo Kielmann. 2010. Bag-of-tasks scheduling under budget constraints. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 351–359.

[11] Kassian Plankensteiner, Radu Prodan, Thomas Fahringer, Attila Kertész, and Péter Kacsuk. 2008. Fault Detection, Prevention and Recovery in Current Grid Workflow Systems. In *Grid and Services Evolution, Proceedings of the 3rd CoreGRID Workshop on Grid Middleware, June 5-6, 2008, Barcelona, Spain*. 1–13.

[12] Prateek Sharma, Stephen Lee, Tian Guo, David E. Irwin, and Prashant J. Shenoy. 2015. SpotCheck: designing a derivative IaaS cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. 16:1–16:15.

[13] Supreeth Subramanya, Tian Guo, Prateek Sharma, David E. Irwin, and Prashant J. Shenoy. 2015. SpotOn: a batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. 329–341.

[14] Long Thai, Blesson Varghese, and Adam Barker. 2014. Executing Bag of Distributed Tasks on the Cloud: Investigating the Trade-Offs between Performance and Cost. In *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014*. 400–407.

[15] Long Thai, Blesson Varghese, and Adam Barker. 2018. A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds. *Future Generation Comp. Syst.* 82 (2018), 1–11.

[16] Johan Tordsson, Rubén S Montero, Rafael Moreno-Vozmediano, and Ignacio M Llorente. 2012. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future generation computer systems* 28, 2 (2012), 358–367.

[17] Ji Wang, Weidong Bao, Xiaomin Zhu, Laurence T. Yang, and Yang Xiang. 2015. FESTAL: Fault-Tolerant Elastic Scheduling Algorithm for Real-Time Tasks in Virtualized Clouds. *IEEE Trans. Computers* 64, 9 (2015), 2545–2558.

[18] Min Yao, Peng Zhang, Yin Li, Jie Hu, Chuang Li, and Xiang-Yang Li. 2014. Cutting Your Cloud Computing Cost for Deadline-Constrained Batch Jobs. In *2014 IEEE International Conference on Web Services, ICWS, 2014, Anchorage, AK, USA, June 27 - July 2, 2014*. 337–344.

[19] Qin Zheng, Bharadwaj Veeravalli, and Chen-Khong Tham. 2009. On the Design of Fault-Tolerant Scheduling Strategies Using Primary-Backup Approach for Computational Grids with Low Replication Costs. *IEEE Trans. Computers* 58, 3 (2009), 380–393.